5
Canvases and Openwin

Perhaps the most important object in the XView Toolkit is a *canvas*—the area in which an application displays graphics and handles input. The canvas object is similar to that of a painter's canvas. The artist's painting is drawn onto the canvas and the canvas is mounted in a frame. The canvas may be larger than the frame, but the person looking at the canvas only sees what is within the boundaries of the frame. If the canvas is larger than what is viewable, the painting can be moved around making different portions of the canvas visible.

An XView canvas object allows the user to view a graphic image that is too large for the window or even the display screen. The viewable portion of the graphic image is part of the *viewport* or *view window* of the image. Many different views of the image can use the same canvas object. While each view maintains its own idea of what it is displaying, the canvas object manages all the view windows as well as the graphic image that all views share. The ability for the canvas to maintain different views of the graphic image is a property that is inherited from the canvas's superclass, the `OPENWIN` package. These properties provide for *splitting* and *scrolling* views. You cannot create a canvas object with multiple views; views are split and joined generally by the user via the attached scrollbars. It is possible to programmatically split and scroll views, but OPEN LOOK's interface specification indicates that scrollbars provide the ability to split views. When a view is split, each new view may be further split into two more views, and so on. All the views are still a part of the same canvas object.

The `OPENWIN` package is an example of a *hidden* class. You cannot instantiate an openwin object independently from canvas or text subwindows. Figure 5-1 shows the openwin and canvas object hierarchy.

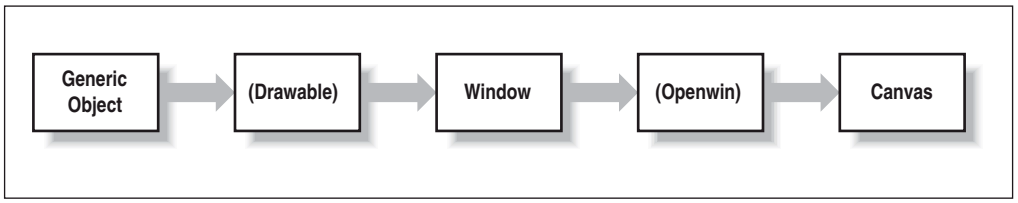

Figure 5-1. Canvas class hierarchy

Chapter 8, *Text Subwindows*, contains information about text subwindows. The canvas object is different from the text object in that it maintains an image that can be manipulated by the user. The openwin object, and thus the canvas object, are broken down into three parts: the *main subwindow*, the *view window*, and the *paint window*.

Each view displays a portion of a corresponding paint window. The paint window need not be the size of the corresponding view or the canvas subwindow. Figure 5-2 shows an example of one canvas object providing separate views into one graphic image. What each view displays is independent of what the other views display. A view may even display a portion of an image that is currently displayed in another view.

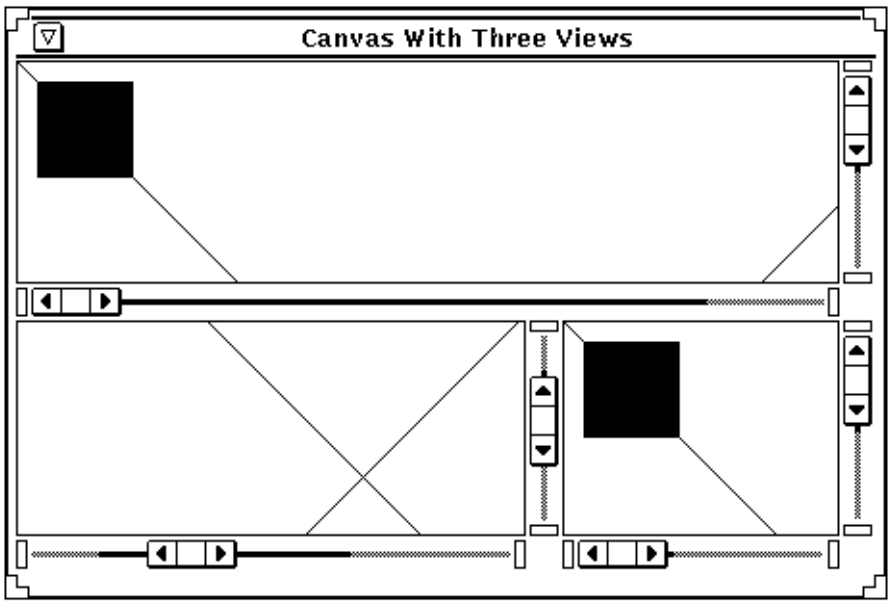

Figure 5-2. A canvas subwindow with multiple views

5.1 Canvas Model

The components of a canvas subwindow and their relationships can be seen in Figure 5-3. To summarize, three types of windows are involved with the canvas object:

Canvas Subwindow — Owned by a frame and manages one or more views. The canvas is subclassed from the `OPENWIN` package so all Openwin attributes must be set to the instance of the canvas object.

View Window — Represents the visible portion of the paint window—whenever the paint window associated with a view window changes, it is reflected in the view window. If there is more than one view window, the views are tiled. Vertical and/or horizontal scrollbars can be attached to the view subwindow to allow the user to modify which portion of

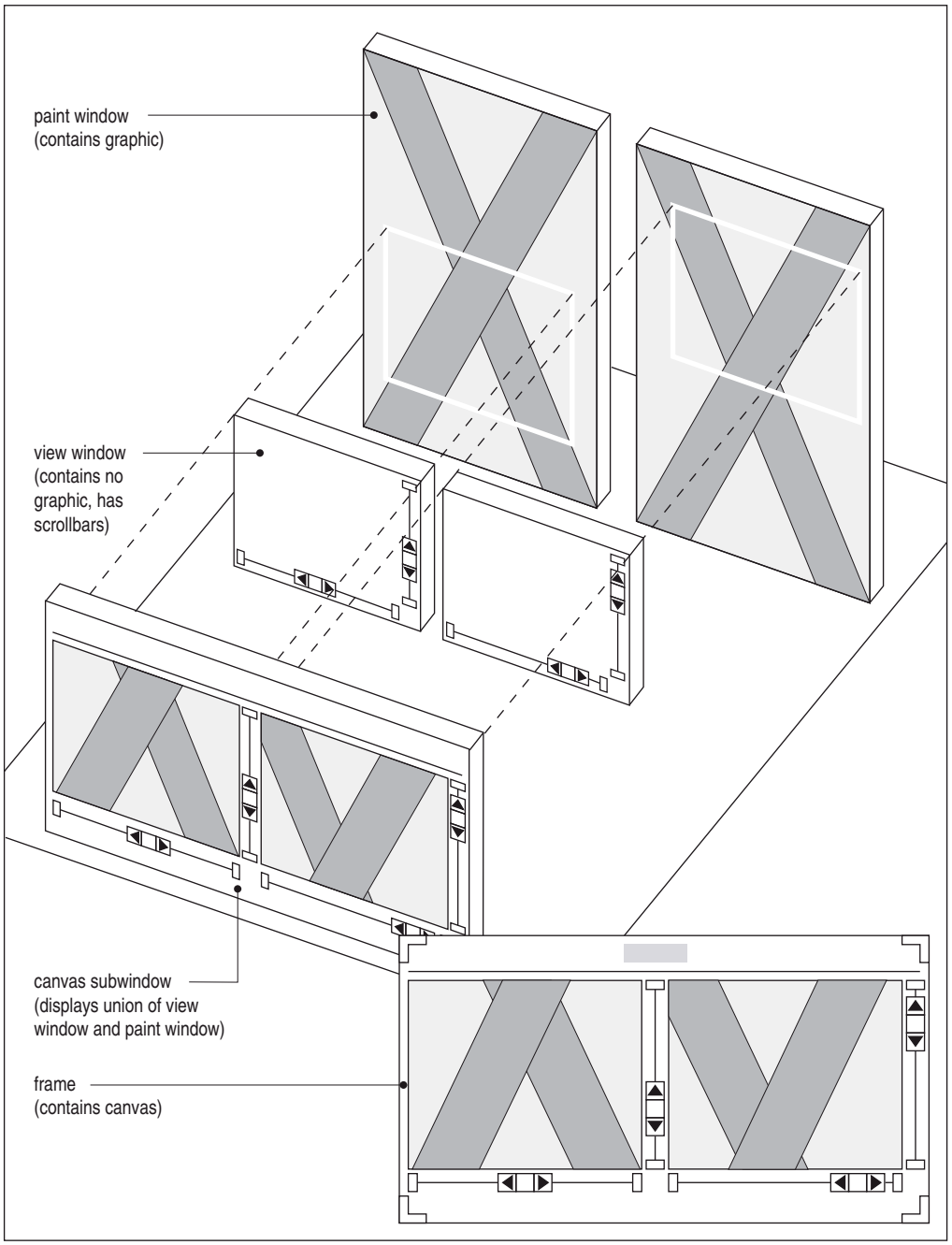

Figure 5-3. Canvases, views, and paint windows

the paint window is displayed for that particular view. The size of the view window can vary among all the views. Only views can be split. No graphics or user events take place in this window.

Paint Window — Graphics and events (mouse/keyboard) take place in the paint window. There is one paint window per view window. All paint windows in the canvas are the same size regardless of the size of the canvas or of the corresponding view windows. When a view is split, the old view reduces in size and a new view is created. With the new view, a new paint window is created that is identical to the paint window from the old view. This includes the same visual, width, height, depth, and graphic image. However, callback functions and event masks are *not* inherited and *must* be manually installed in all new paint windows.

5.2 Creating a Canvas

The CANVAS package is defined in the header file *<xview/canvas.h>* so programs that use canvases must include this file. This header file includes the OPENWIN package automatically. Like all objects in XView, a canvas is created with xv_create():

```
Canvas canvas;

canvas = (Canvas)xv_create(owner, CANVAS, attrs);
```

Here, xv_create() returns a handle to a new canvas subwindow. The owner of a canvas must be a FRAME object. All three subwindows of the canvas are created at this point: the canvas subwindow, the view window and the paint window. By using this syntax, the attributes of the canvas default to those set for the CANVAS package plus any attributes that may be inherited from the owner of the canvas.

The windows in the canvas object inherit many of their attributes from the canvas's parent, screen, or display (depending on the window property). However, some attributes are set or reset explicitly. For example, the attribute CANVAS_RETAINED, which controls whether the server should retain windows, is turned on.* Toggling this attribute causes the canvas to cycle through all of the paint windows and change their WIN_RETAINED attribute. Also, the window has its BitGravity set to NorthWestGravity by default. This value is set by the attribute CANVAS_FIXED_IMAGE. If TRUE, then the BitGravity property on the paint window is set to NorthWestGravity. If FALSE, BitGravity is set to ForgetGravity. This is discussed in more detail in Section 5.3, "The Repaint Procedure."

The width and height of the paint window and the view window default to the size of the canvas when it is realized. Unless otherwise specified, those sizes are governed by the object that is the owner of the canvas.

*CANVAS_RETAINED does not affect the view windows, which are not retained.

5.2.1 Drawing in a Canvas

The use of canvases implies that your application wants to display a graphic image that the user can either manipulate or generate. In conventional (not server-client based) windowing systems, when you request to create a window, you usually get a window back that you can draw into. This is not exactly true for X. Although you get a window back from the request, you are not guaranteed to be able to draw into it until it has been successfully mapped (displayed) on the screen. In the general case for XView, this does not happen until `xv_main_loop()` is called. Therefore, you should *not* do the following:

```
canvas = (Canvas)xv_create(frame, CANVAS, NULL);
win = (Window)xv_get(canvas_paint_window(canvas), XV_XID);

XDrawString(dpy, win, gc, x, y, "Hello World.", 11);
```

Instead, you should design your program such that your repaint routine knows exactly what the contents of the canvas should be so that it can reproduce the image. Here are two helpful hints for typical applications:

- *Use your repaint proc*. Never call any graphics routines before `xv_main_loop()` is called. Routines that draw anything into windows should be called directly or indirectly from your canvas repaint procedure or event handler.
- *Use internal data*. The repaint routine should be able to repaint a canvas window based on some sort of internal data. To maintain system performance, the data should be in core; it should *not* be on disk (in a file), from a network connection or from interaction with the user. If the data is received from those media, the data should already have been updated by the time the repaint routine is called.

To give you an idea of the issues involved here, we'll look at several common applications that typically use canvases.

5.2.1.1 Draw programs

Draw programs are usually applications that maintain a *display list* of geometric shapes such as lines, circles, rectangles and so on. The user generates these shapes using the mouse or keyboard. The canvas's `WIN_EVENT_PROC` procedure handles mouse and keyboard input from the user, and the user interface generally describes the shape that is currently being drawn.

When the user initiates mouse clicks, drags or keyboard actions, the event handler picks up these events and is fully expected to modify the canvas window accordingly (e.g., by "rubber banding" the object). Upon receipt of the appropriate event (button release, perhaps), the event handler adds the new geometric item to the display list.

In order to reconstruct what the canvas should be displaying, the repaint routine references the updated display list. Obviously, the display list should contain enough information in it to be able to tell the repaint routine what colors to use, line thickness, and so on.

5.2.1.2 Paint programs

Paint programs are *pixel based*, meaning that the image that the user manipulates is typically a bitmap (monochrome) or a color pixmap. In this case, the pixmap is used as the internal data. If the user uses a *brush* to modify the image, the modification to the canvas window is handled in the event procedure as above, but the image that the user is editing is also updated so that when the repaint routine is called, it can use a function such as `XCopyArea()` to copy the portion of the image onto the canvas window. This is also a case where the event handler may draw directly onto the window.

If the program starts up and already has an image to work with (e.g., the user requests to load a previously saved image that was stored into the file), *do not render the image onto the paint window before* `xv_main_loop()` *is called.* Eventually, the repaint procedure will be called and the image should be rendered at that time.

5.2.1.3 Text-based programs

If you want to display text, as in desktop publishing packages, or if you just want a simple program that displays a window with text, then the same thing applies as with the previous examples: render the text in the repaint routine based on internal data (e.g., a text string or set of strings). Again, the data must contain enough information to allow your repaint routine to repaint the text as it was intended (e.g., font type, style, size, color).

Event-handling routines that accept keyboard input may render the new text directly to the canvas window, but the text entered should also be saved internally for the benefit of the repaint routine.

5.2.1.4 Visualization programs

It is common to have an application that displays the time, network traffic, CPU usage, file system integrity, or the current Dow Jones Industrial Averages. Such applications get their data from a variety of sources such as the system clock, UNIX sockets, the file system or the output of another application that has been forked. In the past, such applications might have been written where the repaint routine accesses the information. This model does not apply in a networked windowing system because of its asynchronous nature; the window system and the application may not be in sync. A program should have a separate method for retrieving data apart from the repaint procedure. See the program *animate.c* in Chapter 20, *The Notifier*, for an example.

In many of these cases (with the possible exception of the *meters*), if the application is well designed, the repaint routines and the event handlers may be calling the same internal routines which render graphics. Therefore, when writing functions that draw into a canvas, you should consider the generic case where the function could be called from a repaint routine, event handler or anywhere else. The *canvas_event.c* shown in Example 5-3 calls the repaint routine directly from the event handler.

5.2.1.5 Rendering graphics

The preferred form of rendering graphics from an XView application is to use Xlib graphics calls. Volume One, *Xlib Programming Manual*, has a complete discussion of Xlib graphics programming. Throughout this book, you will find examples of drawing into canvases using Xlib graphics routines. Appendix F, *Example Programs*, has several longer programs that demonstrate Xlib graphics. The XView graphics model, which is available, is almost identical to the SunView model for graphics and is provided for backwards compatibility with SunView. Because XView graphics calls are wrappers to the underlying Xlib calls, these functions are not recommended for graphics-intensive applications or for use by programmers who are not already familiar with SunView.

5.3 The Repaint Procedure

It is always the responsibility of the application to repaint its canvas at any time. Even though there may be a retained canvas that the X server maintains, there is no guarantee that there will be enough memory for the server to maintain it. For this reason, all canvases should install a routine to handle repainting.

To install a repaint procedure, use the attribute `CANVAS_REPAINT_PROC` and specify a callback function as its value:

```
extern void my_repaint_proc();
...
canvas = (Canvas)xv_create(frame, CANVAS,
    ...
    CANVAS_REPAINT_PROC,    my_repaint_proc,
    ...
    NULL);
```

The repaint routine installed is called any time all or a portion of the canvas needs to be re-displayed. This always happens when the canvas is mapped on the screen for the first time (causing an `Expose` event). If the canvas is *not* retained, the repaint procedure is called when:

- The canvas is resized.
- The canvas has been moved in front of obscuring windows.
- The user uses the scrollbar to render a different part of the paint window visible.

If the canvas *is* retained and has not changed size, the server refreshes the window without calling the repaint routine. This includes all exposures except for those that are the result of a resize of the window or the initial mapping of the canvas onto the screen. However, if the canvas is not retained, the repaint routine is called in all of these cases.

The repaint callback routine will be called once for each view the canvas is maintaining. If the canvas has been split several times, then the repaint routine will get called for each view that needs repainting. One of the parameters to the callback routine is a variable that describes the region that has been exposed or needs repainting. When a window is initially

displayed on the screen, the exposed region is the entire canvas. However, this area may not be a contiguous area of the window. For example, as shown in Figure 5-4, if a window that is partially obscured by two windows is brought forward, two separate areas are exposed.

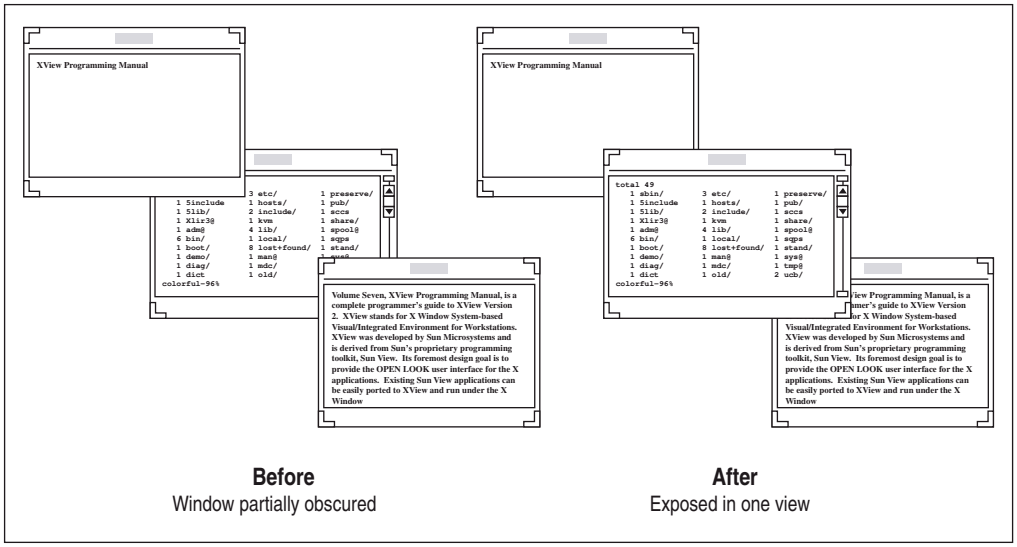

Figure 5-4. Window before and after an Expose event

The attribute `WIN_COLLAPSE_EXPOSURES` governs how many times the repaint routine is called. By default, XView collapses `Expose` (and `GraphicsExpose`) X events destined for the same window; that is, XView waits for the exposure count member to reach zero. After all exposure X events have arrived (when `count == 0`), XView generates a `WIN_REPAINT` event and calls the repaint procedure. This `WIN_REPAINT` represents all the areas in the window that have been damaged (it can be a disjoint set of rectangles). The X event associated with the `WIN_REPAINT` event (accessed through the `event_xevent()` macro) represents a bounding rectangle of all the damaged areas in the window (basically the union of all the damaged areas in the window).

However, sometimes the application wants to monitor the incoming `Expose` (and `GraphicsExpose`) X events, monitoring piece by piece the count member in the `Expose` X event itself. This can be done by setting `WIN_COLLAPSE_EXPOSURES` to `FALSE` on the canvas's paint window. As exposures come into a window with this attribute set to `FALSE`, they will be immediately sent to the client's repaint procedure. The client will thus receive several `WIN_REPAINT` events, all for the same window. The area of exposure is set to each region as it is exposed.

By default, the repaint routine is called once per window exposed. However, there may be situations where there are more windows exposed in the same canvas object. For example, in Figure 5-5, two view windows have been exposed as a result of bringing the window to the top of the window stack. In this case, the repaint routine will be called twice—once for the

paint window of each view exposed. Only one area of each view window is exposed, so the value of the WIN_COLLAPSE_EXPOSURES attribute does not apply.

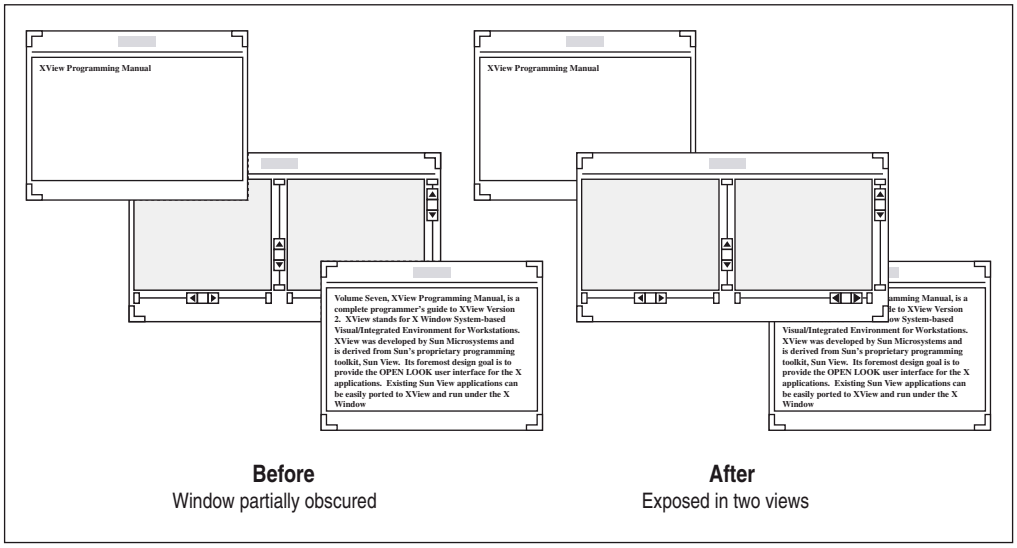

Figure 5-5. Window with two views before and after an Expose event

If at any time you need to get the entire viewable area of the canvas, or more specifically, of an arbitrary paint window within the canvas, you can use the attribute CANVAS_VIEWABLE_RECT:

```
Rect *rect;
Xv_Window pw = canvas_paint_window(canvas);

rect = (Rect *)xv_get(canvas, CANVAS_VIEWABLE_RECT, pw);
```

The rect pointer returned points to an internal data structure that describes the viewable area of the paint window specified. This structure changes for each call, so if the value is to be retained, it should be copied.

Before the repaint routine is called, the window can be cleared in one of two ways. If CANVAS_AUTO_CLEAR is set to TRUE, then the paint window being repainted is automatically cleared. CANVAS_AUTO_CLEAR is really defined as OPENWIN_AUTO_CLEAR since this is a property of the OPENWIN package. Automatically clearing the window happens any time the window needs repainting. This means that the exposed area represents the entire window. If this attribute is set to FALSE (the default), the repaint routine should prepare to clear all, or portions of, the window that needs to be repainted. The contents of the window in the exposed areas is undefined. If you are going to repaint those areas opaquely (that is, leave no transparent portions), then you do not need to clear the area. However, if any transparent portion of the area will be repainted (e.g., if your gc.function is set to GXxor), you should clear the window first by using XClearArea() or XClearWindow().

Alternatively, the window may be cleared automatically by the Xlib internals if the window has actually changed size. The BitGravity attribute for the window controls whether the data in the window is cleared or just moved around to different locations of the window according to its new size. If BitGravity is set to ForgetGravity, then the data in the window is discarded, resulting in the window getting cleared and the canvas's repaint procedure getting called. As mentioned before, this value can be set by setting the attribute CANVAS_FIXED_IMAGE to FALSE. But to have more direct control over the BitGravity of the window, the window attribute WIN_BIT_GRAVITY may be set to any of the legal values provided by Xlib (ForgetGravity, NorthGravity, NorthWestGravity and so on). Section 4.3.3, "Bit Gravity," in Volume One, *Xlib Programming Manual*, discusses this in full detail. Note: this should not be confused with WIN_WINDOW_GRAVITY which controls the reposition of subwindows when a parent window is resized. This task is left to the FRAME package, since it controls subwindow layout.

If the attribute CANVAS_CMS_REPAINT is set to TRUE, the repaint procedure is called automatically whenever a new colormap segment is set on the canvas or the foreground and background colors of the canvas are changed using WIN_FOREGROUND_COLOR and WIN_BACKGROUND_COLOR.

The parameters to the repaint routine provide information about which window and which areas within the window need to be repainted.

The repaint procedure takes one of two different forms:

```
void
repaint_proc(canvas, paint_window, repaint_area)
    Canvas        canvas;
    Xv_Window     paint_window
    Rectlist     *repaint_area;
```

or:

```
void
repaint_proc(canvas, paint_window, dpy, xwin, area)
    Canvas        canvas;
    Xv_Window     paint_window
    Display      *dpy;
    Window        xwin;
    Xv_xrectlist *area;
```

The routine takes the first or second form depending on the value of the attribute CANVAS_X_PAINT_WINDOW. If FALSE, the first, simpler form of the repaint procedure is called. If TRUE, the repaint routine gets passed the parameters shown in the second form. The second method is more useful since it saves you from writing code for extracting the Display and the XID of the paint window.

In both forms, the first two parameters are rather obvious—the paint_window is the paint window associated with the canvas that needs repainting (not the view window). The paint_window contains an X window whose XID can be gotten by using xv_get():

```
Window xwin = (Window)xv_get(paint_window, XV_XID);
```

xwin is set to the actual X window referenced by the paint window. This is the way to obtain the XID from the paint window because the first form of the repaint procedure (when CANVAS_X_PAINT_WINDOW is FALSE) does not provide a handle for you.

In the first form of the repaint procedure, the only other parameter is the `repaint_area`. This is a linked list of rectangular areas in the paint window that have been exposed and need to be repainted. The `Rectlist` type is a linked list of `Rect`'s as shown in *<xview/rect.h>*. The declaration for the `Rect` type is:

```
typedef struct rect {
    coord r_left, r_top;
    short r_width, r_height;
} Rect;
```

The type `coord` is `#define`'d as type `short`. The `Rectlist` is declared in *<xview/rectlist.h>* as:

```
typedef struct  rectnode {
   struct rectnode *rn_next;  /* Pointer to next rectnode */
   struct rect      rn_rect;
} Rectnode;

typedef struct rectlist {
   coord           rl_x, rl_y; /* Offset to apply to each rect
                                * in list including bound */
   struct rectnode *rl_head;    /* Pointer to first rectnode */
   struct rectnode *rl_tail;    /* Pointer to last rectnode */
   struct rect      rl_bound;   /* Describes bounding rect of
                                * all rects in list */
} Rectlist;
```

The `repaint_area` parameter is of type `Rectlist`, so the application has several ways it can approach repainting the window. It could just ignore the parameter and repaint the entire window, or it could repaint the entire area described by the `rl_bound` field of the `rectlist` structure or it could loop through all the `Rectnode`'s and repaint those areas individually. Deciding which method to choose should be based on how complicated the redrawing is.

When `CANVAS_X_PAINT_WINDOW` is set to `FALSE`, the internal clipping for the paint window is set to be the same region or regions described by the repaint parameter of the repaint routine. In most cases, this means that `OPENWIN_AUTO_CLEAR` will only clear those areas—not the entire window.* The default behavior may be overridden by setting `WIN_NO_CLIPPING` to `TRUE` for the *canvas*. Setting this attribute cycles through all the paint windows in the canvas and sets the window property `WIN_NO_CLIPPING`. Having the clipping on the window turned off means that `OPENWIN_AUTO_CLEAR` will cause the entire window to be cleared.†

By setting the attribute `CANVAS_X_PAINT_WINDOW` to `TRUE`, the repaint routine gets passed the extra X-specific parameters as shown in the second form of the repaint procedure. These extra parameters include handles to the `Display` and the X window of the `paint_window`.

*For those using the SunView-compatible drawing routines such as `pw_vector()`, rendering is clipped to the area or areas described by `repaint_area`.
†This also affects the SunView-compatible routines—`pw_vector()` will not be clipped.

Example 5-1 shows how a repaint procedure might be used.

Example 5-1. The line.c program

```
/*
 * line.c -- demonstrates installing a repaint routine in a canvas.
 * The routine is called whenever the canvas needs to be repainted.
 * This usually occurs when the canvas is exposed or resized.
 */
#include <X11/Xlib.h>
#include <xview/xview.h>
#include <xview/canvas.h>
#include <xview/xv_xrect.h>

main(argc, argv)
int argc;
char *argv[ ];
{
    Frame frame;
    void  canvas_repaint_proc();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(NULL, FRAME, NULL);

    (void) xv_create(frame, CANVAS,
        CANVAS_REPAINT_PROC,    canvas_repaint_proc,
        CANVAS_X_PAINT_WINDOW,  TRUE,
        NULL);

    xv_main_loop(frame);
}

/*
 * repaint routine draws a line from the top left to the bottom right
 * corners of the window
 */
void
canvas_repaint_proc(canvas, paint_window, dpy, xwin, xrects)
Canvas        canvas;         /* unused */
Xv_Window     paint_window;   /* unused */
Display      *dpy;
Window        xwin;
Xv_xrectlist *xrects;         /* unused */
{
    GC gc;
    int width, height;

    gc = DefaultGC(dpy, DefaultScreen(dpy));
    width = (int)xv_get(paint_window, XV_WIDTH);
    height = (int)xv_get(paint_window, XV_HEIGHT);

    XDrawLine(dpy, xwin, gc, 0, 0, width, height);
}
```

Because the program uses Xlib calls (and thus, the repaint routine is being passed different parameters), the header files *<X11/Xlib.h>* and *<xview/xv_xrect.h>* must be added at the top of the program. The parameters `dpy` and `xwin` are handles to the X `Display` and the X `Window`, respectively. The `GC` (graphics context) is taken from the default `GC` of the screen.

The `xrects` parameter represents the exposed, or "damaged," region of the paint window. The `Xv_xrectlist` is declared in *<xview/xv_xrect.h>* as:

```
#define XV_MAX_XRECTS 32
typedef struct {
   XRectangle   rect_array[MAX_XRECTS];
   int          count;
} Xv_xrectlist;
```

When the canvas is first displayed, this region will be the entire paint window (or the portion that is viewable by the view window). But in cases where another window that has partially obscured it moves away, perhaps only a portion of the paint window will need repainting. Therefore, the `xrects` variable can aid in setting the clip rectangles in the `GC`, as shown in Example 5-2.

Example 5-2. Repainting objects within a damaged region

```
/* canvas_repaint_proc()
 *
 * Draws onto the canvas using Xlib drawing functions.
 *
 * Uses the current clipping rectangle to:
 *  1. Restrict graphics output by setting the
 *     clip_mask in the graphics context.
 *  2. Do "smart repainting" by only painting the objects
 *     that lie within the damaged region (not being done
 *     in this example).
 */
void
repaint_proc(canvas, paint_window, display, xid, xrects)
    Canvas canvas;
    Xv_Window paint_window;
    Display *display;
    Window xwin;
    Xv_xrectlist *xrects;
{
    extern GC gc;
    int width, height;

    width = (int)xv_get(paint_window, XV_WIDTH);
    height = (int)xv_get(paint_window, XV_HEIGHT);

    /*
     * Set clip rects, if any
     */
    if (xrects)
        XSetClipRectangles(display, gc, 0, 0, xrects->rect_array,
            xrects->count, Unsorted);
    else {
        XGCValues gc_val;

        gc_val.clip_mask = None;
```

Example 5-2. Repainting objects within a damaged region (continued)

```
        XChangeGC(display, gc, GCClipMask, &gc_val);
    }

    XDrawLine(display, xwin, gc, 0, 0, width, height);
}
```

Because this routine sets the clip mask of the GC, we want to be sure that we *do not use the default* GC of the screen as we did in Example 5-2 or it will interfere with other programs (such as the window manager). The GC shown here is declared as `extern`, assuming that the application has created it somewhere else using `XCreateGC()`.

5.4 Controlling Canvas Sizes

The size of the canvas subwindow is usually determined by the frame window. Thus, the canvas changes as the user resizes the frame. Applications largely concern themselves with the size of the paint window.* The paint window does not affect the size of the viewable canvas, but the viewable portion of the paint window is important.

Although the width and height of the canvas subwindow can be set explicitly, unless done so, the default size of the subwindow and the paint window is determined by the parent frame. If the frame resizes, the canvas object resizes proportionally according to how the frame chooses to resize the canvas. If several other windows (canvases, panels, whatever) are in the frame, the frame might choose to lay out and size those subwindows differently (according to available and required space from other windows). The canvas window itself, as well as all window objects, can be sized using XV_WIDTH and XV_HEIGHT.

5.4.1 Automatic Canvas Sizing

The paint window's size may fluctuate with that of the canvas subwindow's size. The attributes CANVAS_AUTO_EXPAND and CANVAS_AUTO_SHRINK maintain the relation of the canvas subwindow and paint window in the event of any kind of window resizing. Both of these attributes default to TRUE, allowing the paint window to always correspond to the size of the canvas subwindow. If the canvas subwindow becomes larger, the paint window size changes to that size. If the frame changes size, the canvas subwindow changes size and so does the paint window. This happens regardless of how many view windows there are. The size of view windows does not affect the size of the paint window.

Specifically, if CANVAS_AUTO_EXPAND is TRUE, then the width and height of the paint window cannot be less than that of the canvas subwindow. Setting the attribute CANVAS_AUTO_EXPAND allows the paint window to grow bigger as the user stretches the window. If a resize of the subwindow occurs such that the size of the paint window is less than the size of the canvas subwindow, the paint window is expanded to be at least that size.

*There may be more than one paint window to a canvas; but all paint windows in a canvas are the same size, so it's a moot point.

Conversely, if the canvas subwindow's size shrinks, then the paint canvas size does not change because its size is already greater than or equal to the size of the canvas subwindow—no expansion is necessary.

If CANVAS_AUTO_SHRINK is TRUE, the canvas object checks that width and height of the paint window are not greater than that of the canvas subwindow. Setting CANVAS_AUTO_SHRINK forces the paint window to grow smaller as the size of the canvas subwindow gets smaller. If the user resizes the frame such that the canvas subwindow is smaller than the size of the paint window, then the paint window is reduced to the size of the new subwindow.

You can also set a minimum width and height for the canvas using the attributes CANVAS_MIN_PAINT_WIDTH and CANVAS_MIN_PAINT_HEIGHT. Regardless of whether or not CANVAS_AUTO_SHRINK is set to TRUE or FALSE, the attributes CANVAS_MIN_PAINT_WIDTH and CANVAS_MIN_PAINT_HEIGHT impose the minimum CANVAS_WIDTH and CANVAS_HEIGHT respectively.

5.4.2 Explicit Canvas Sizing

The attributes CANVAS_WIDTH and CANVAS_HEIGHT can be set to establish the size of the *paint window*. Automatic sizing should be turned off. Otherwise, as soon as the canvas window is realized, the paint window may be automatically resized to the new dimensions. This all depends on whether either or both of the auto-expand or auto-shrink attributes are set. The following code fragment shows that one can be set and the other unset for specific needs:

```
Canvas canvas;

canvas = (Canvas)xv_create(frame, CANVAS,
    CANVAS_AUTO_SHRINK,  FALSE,
    CANVAS_AUTO_EXPAND,  TRUE,
    CANVAS_WIDTH,        100,
    CANVAS_HEIGHT,       200,
    NULL);
```

With these settings, the paint window will initially be set to 100 by 200. If the subwindow is realized at a larger size, the canvas will be expanded to the new dimensions. That is, if the frame in which the canvas resides is larger, it may affect the initial size of the paint window. However, if the canvas is realized or resized at smaller dimensions, the canvas will retain its original size. In short, these settings will force the paint window to grow to the maximum size that the window will ever be—it will never shrink. In typical usage, you would set the auto-expand and auto-shrink attributes to FALSE and explicitly set CANVAS_WIDTH and CANVAS_HEIGHT. Alternatively, you would not initialize the width and height and set both CANVAS_AUTO_EXPAND and CANVAS_AUTO_SHRINK to TRUE. A draw program might allow the paint window to be sized automatically, since the display list of geometric objects is the underlying feature of the program. However, a paint program would set explicit width and height attributes of the graphic, disallowing any resizing of that graphic.

The following code fragment creates a canvas with a fixed-size paint window that is not affected by resizing:

```
Canvas canvas;

canvas = (Canvas)xv_create(frame, CANVAS,
    CANVAS_AUTO_SHRINK,  FALSE,
    CANVAS_AUTO_EXPAND,  FALSE,
    CANVAS_WIDTH,        1000,
    CANVAS_HEIGHT,       1000,
    NULL);
```

This call sets the initial size of the paint window to 1000 by 1000 pixels. The origin of the paint window's coordinate system is the upper-left corner (0,0) and the lower-right corner (CANVAS_WIDTH-1, CANVAS_HEIGHT-1). Note that we did not set the size of the canvas subwindow. Instead, we allowed it to be determined by the frame size. The size of the paint window remains constant regardless of how the frame and canvas subwindow is resized. If the frame or the canvas subwindow resizes, the subwindow merely changes its view of the underlying paint window, which remains constant.

In the following code fragment, we set the size of the canvas subwindow, using generic attributes:

```
Canvas canvas;

canvas = (Canvas)xv_create(frame, CANVAS,
    CANVAS_AUTO_SHRINK,  FALSE,
    CANVAS_AUTO_EXPAND,  FALSE,
    CANVAS_WIDTH,        1000,
    CANVAS_HEIGHT,       1000,
    XV_WIDTH,            200,
    XV_HEIGHT,           100,
    NULL);
```

Here, a canvas subwindow is created that is 200 pixels wide and 100 pixels high. All other attributes about this canvas object are the same as the previous example: the paint window is going to be 1000x1000 in width and height. The problem with this canvas is that the user has no way to view different parts of the paint window. To handle that, scrollbars should be attached to the canvas to provide scrolling. See Section 5.5, "Scrolling Canvases."

5.4.3 Tracking Changes in the Canvas Size

In the event that the canvas paint window has been resized, the program has the opportunity to track this event by installing a callback routine. This routine is installed using CANVAS_RESIZE_PROC. The client's resize procedure is called only when the width or height of the canvas's paint window changes. Its form is:

```
void
sample_resize_proc(canvas, width, height)
    Canvas canvas;
    int    width;
    int    height;
```

The parameters to the resize procedure are the canvas, and the width and height of the canvas.

If you need to handle resize events for the canvas or the view windows, provide an event handler for those windows, using the attributes WIN_EVENT_PROC and WIN_CONSUME_EVENTS (see Section 5.7, "Handling Input in the Canvas Package," for more information).

5.5 Scrolling Canvases

Many applications need to view and manipulate a large object through a smaller viewing window. To facilitate this, packages that are subclassed from the openwin class may have scrollbars attached to their subwindows.

The following code fragment creates a canvas that can be scrolled in both directions:

```
Canvas     canvas;
Scrollbar  h_scrollbar, v_scrollbar;

canvas = (Canvas)xv_create(frame, CANVAS,
    CANVAS_AUTO_EXPAND,  FALSE,
    CANVAS_AUTO_SHRINK,  FALSE,
    CANVAS_WIDTH,        1000,
    CANVAS_HEIGHT,       1000,
    NULL);

h_scrollbar = (Scrollbar)xv_create(canvas, SCROLLBAR,
    SCROLLBAR_DIRECTION, SCROLLBAR_HORIZONTAL,
    NULL);

v_scrollbar = (Scrollbar)xv_create(canvas, SCROLLBAR,
    SCROLLBAR_DIRECTION, SCROLLBAR_VERTICAL,
    NULL);
```

Because the SCROLLBAR package is being used here, the header file <*xview/scrollbar.h*> must be included. Chapter 10, *Scrollbars*, discusses scrollbars more completely and also gives further examples of how to scroll canvases.

The owner of the scrollbars is the canvas so that the scrollbars are automatically attached to the canvas's view. If the user scrolls the canvas, your canvas's repaint procedure will be called provided that the canvas's WIN_RETAINED attribute is set to FALSE. This is important because setting WIN_RETAINED to TRUE assumes that you are not interested in handling repainting for scrolling. In other words, as long as the user does not do anything that changes the contents of the image, you do not need to be informed when the user scrolls the image. If you want to be informed of scrolling, set WIN_RETAINED to FALSE and your repaint routine will be called with the *exposed area* parameter describing the new area that just scrolled into view.* If there are many views in the canvas, the paint window associated with the view that scrolled is in the second parameter to the repaint function: the paint_window.

*The exposed area passed to the repaint procedure is of type Xv_xrectlist if CANVAS_X_PAINT_WINDOW is set to TRUE or Rectlist if CANVAS_X_PAINT_WINDOW is set to FALSE.

5.6 Splitting Canvas Views

There are two methods by which the application may split the views of a canvas (or any openwin-classed object). The first method is for the user to use the scrollbars to split views. This method is more common and complies with the OPEN LOOK specification. The alternate method is for the application to make calls to `xv_set()` with attribute-value pairs that tell where and how a view should be split.

Whenever views are split, the following attributes are propagated from the split paint window to the new paint window:

- `WIN_BACKGROUND_COLOR`
- `WIN_FOREGROUND_COLOR`
- `WIN_CMS`
- `WIN_COLUMN_GAP`
- `WIN_COLUMN_WIDTH`
- `WIN_CURSOR`
- `WIN_EVENT_PROC`
- `WIN_ROW_GAP`
- `WIN_ROW_HEIGHT`
- `WIN_X_EVENT_MASK`

5.6.1 Splitting Views Using Scrollbars

To set up the canvas so that the user can split it using scrollbars, the canvas should have scrollbars attached as shown in the previous example with the additional attribute `SCROLLBAR_SPLITTABLE` set to `TRUE`:

```
h_scrollbar = (Scrollbar)xv_create(canvas, SCROLLBAR,
    SCROLLBAR_DIRECTION,    SCROLLBAR_HORIZONTAL,
    SCROLLBAR_SPLITTABLE,   TRUE,
    NULL);

v_scrollbar = (Scrollbar)xv_create(canvas, SCROLLBAR,
    SCROLLBAR_DIRECTION,    SCROLLBAR_VERTICAL,
    SCROLLBAR_SPLITTABLE,   TRUE,
    NULL);
```

With this attribute set, the scrollbars have the ability to split view windows in two. The user splits the view by selecting the *cable anchors* at the endpoints of the scrollbars and dragging them towards the center of the scrollbar. A pop-up menu provided with the scrollbars also provides this functionality. The entire view will be split at the point the mouse button is released, provided there is enough room for a new view at that point. Each view can scroll its own underlying paint window independently of other views.

5.6.2 Splitting Views Using xv_set()

Splitting a view by setting attribute-value pairs in the view is a less common method since the scrollbar already provides this functionality. However, there is a programmatic interface for splitting views whether or not those views have scrollbars attached to them.

The attribute OPENWIN_SPLIT is followed by a list of attribute-value pairs that indicate specifically how a view is to be split. Only attributes that are prefixed with OPENWIN_SPLIT may be used in this NULL-terminated list. Other attributes are ignored. The following demonstrates how an arbitrary view window can be split into two parts:

```
Xv_Window view;
view = (Xv_Window)xv_get(canvas, OPENWIN_NTH_VIEW, 0);

xv_set(canvas,
    OPENWIN_SPLIT,
        OPENWIN_SPLIT_VIEW,       view,
        OPENWIN_SPLIT_DIRECTION,  OPENWIN_SPLIT_HORIZONTAL,
        NULL,
    NULL);
```

This very simple example shows that the first *view window* in the canvas will be split horizontally. The place in which the split takes place is, by default, the position of the scrollbar in the view. Assuming the code fragment above, the window is split so that the new view is the same width as the original view but the height is split at the position of the scrollbar. The original view is the remaining height and is on top of the new view.

5.6.3 Getting View Windows

If a canvas has been split several times, resulting in multiple view and paint windows, it is possible to get a handle to a particular view or paint window. This can be done either at the time the view was split or by using xv_get().

5.6.3.1 Getting the newest view

If you want to be notified when the user splits or joins views, you can specify the attribute OPENWIN_SPLIT_INIT_PROC for when the user *splits* a view, and OPENWIN_SPLIT_DESTROY_PROC when the user *joins* a view. These attributes are set in the canvas (or any openwin object). Set these functions by using xv_create() or xv_set() in the following manner:

```
extern void init_split(), join_view();

xv_create(frame, CANVAS,
    ...
    OPENWIN_SPLIT,
        OPENWIN_SPLIT_INIT_PROC,    init_split,
        OPENWIN_SPLIT_DESTROY_PROC, join_view,
    NULL,
    ...
NULL);
```

Write the split and join functions, which take the following parameters:

```
void
init_split(origview, newview, pos)
    Xv_Window origview, newview;
    int pos;

void
join_view(view)
    Xv_Window view;
```

The `pos` parameter above represents the split position, in pixels, of the view. The `origview` and the `newview` parameters represent the view that was originally split and the new resulting view, respectively. These are *not* the paint windows; they are the views themselves. To get a handle to the associated paint window from these views, you can use:

```
Xv_Window paint_window;

paint_window = (Xv_Window)xv_get(view, CANVAS_VIEW_PAINT_WINDOW);
```

Example 6-1 in Chapter 6, *Handling Input*, shows how to handle input in different views.

5.6.3.2 Getting arbitrary views

For each view in an `OPENWIN` object, you can get either the view window or the paint window by choosing either the `CANVAS_NTH_PAINT_WINDOW` or the `OPENWIN_NTH_VIEW` attribute and an integer value for the view window. The first window is 0 and the last window is *n*-1, where *n* is the number of view windows. For instance, to get the second paint window in the canvas, you can use:

```
xv_get(canvas, CANVAS_NTH_PAINT_WINDOW, 1, NULL);
```

You can get the number of available views by calling:

```
int nviews = (int)xv_get(canvas, OPENWIN_NVIEWS);
```

Remember that the number of views corresponds directly to the number of paint windows. Each paint window can be accessed in order, using a simple loop like the following:

```
Xv_Window window;
Canvas    canvas;
int       i = 0;

while (window = (Xv_Window)xv_get(canvas, CANVAS_NTH_PAINT_WINDOW, i)) {
    draw_into_window(window);
    i++;
}
```

The call to `xv_get()` returns `NULL` if you try to get a window number that does not exist (XView does the error checking). Thus, the loop terminates when `xv_get()` returns `NULL`. The value of `i` represents the number of views in the canvas subwindow.

XView provides a pair of macros that facilitate looping through a set of views in a canvas: `CANVAS_EACH_PAINT_WINDOW` and `CANVAS_END_EACH`. The previous loop could be written as:

```
Xv_Window window;
Canvas    canvas;

CANVAS_EACH_PAINT_WINDOW(canvas, window)
    draw_into_window(window);
CANVAS_END_EACH
```

Because the paint windows are different from the view windows, a slightly different method is used for getting view windows:

```
Xv_Window view;
Canvas    canvas;
int       i = 0;

while (window = (Xv_Window)xv_get(canvas, OPENWIN_NTH_VIEW, i)) {
    /* process window */
    i++;
}
```

There is also a macro that loops through all the views in the canvas:

```
Xv_Window view;
Canvas    canvas;

OPENWIN_EACH_VIEW(canvas, view)
    ...
OPENWIN_END_EACH
```

You can get the paint window associated with a view by using the attribute `CANVAS_VIEW_PAINT_WINDOW`:

```
Xv_Window view;
Xv_Window paint_window;

paint_window = (Xv_Window)xv_get(view, CANVAS_VIEW_PAINT_WINDOW);
```

This is useful in situations where you are given the view window and need to get the paint window associated with it. For example, the routines called when views are *split* or *joined* are passed handles to view windows. When a view is split, you will need to get the paint window associated with the new view to install event or repaint callbacks.

5.7 Handling Input in the Canvas Package

This section discusses, to a limited degree, the method for handling and specifying events in a canvas. For a detailed discussion of the types of events used and the proper method for handling them, see Chapter 6, *Handling Input*.

5.7.1 Default Events

The default `canvas_paint_window` event mask is composed of: `KBD_USE`, `KBD_DONE`, `WIN_MOUSE_BUTTONS`, `ACTION_HELP`, and `WIN_ASCII_EVENTS`. Other events may be added to the window event mask by using `xv_set()` and passing the appropriate parameters. The following shows how to enable notification once the Meta key has gone down by enabling `META` events:

```
xv_set(canvas_paint_window(canvas),
    WIN_CONSUME_EVENT, WIN_META_EVENTS,
    NULL);
```

An application that does not need to know about release events can ignore all release events from mouse buttons and keyboard keys, by enabling `WIN_UP_EVENTS` by calling:

```
xv_set(canvas_paint_window(canvas),
    WIN_CONSUME_EVENT, WIN_UP_EVENTS,
    NULL);
```

5.7.2 Notification of Events

In addition to specifying which events the application needs to know about, the program should also install an event callback routine that is called when one of the specified events takes place. The callback routine for event handling is installed using `WIN_EVENT_PROC`. Included are samples that demonstrate how to handle events appropriately using a combination of repaint and event callback routines. However, for a complete discussion of events, you should consult Chapter 6, *Handling Input*, and Chapter 20, *The Notifier*.

The sample program, *canvas_event.c*, in Example 5-3 first creates a base frame. Then it creates a canvas with the attribute `CANVAS_X_PAINT_WINDOW` set to `TRUE` because its repaint procedure (`repaint_proc`) uses Xlib routines to clear the window and draw text strings in the canvas.

Next we specify the events that the application should handle when they occur on the `paint_window`. We are going to listen for keyboard events, pointer motion events and pointer button events. We have not assigned any responses to these events yet; we have just registered them for this window with XView so that the application will be called back if they occur.

We then set the paint window's event handling procedure to be `event_proc`. This is the routine that will decide what to do when the events occur. XView is then started up by calling `xv_main_loop()`, in which event processing starts.

The `event_proc` is called by the Notifier whenever a registered event takes place in any view that has an event handling procedure set. The `event_proc` looks at the type of event it has received and determines the appropriate message to display in the paint window. There is a different message for each type of event that we have registered. There are three message buffers, one each for keyboard events, pointer motion events and pointer button events. After the message buffers are updated, the repaint procedure is called to display them. Note

that we are reusing the `repaint_proc`, instead of writing more code just to display the messages. See Section 5.2.1, “Drawing in a Canvas.”

If the event we have received is of no interest to us, then we return. It is important to do this because the events `WIN_REPAINT` and `WIN_RESIZE` are delivered regardless of the events we have registered with the Notifier. These two events will eventually result in the Notifier calling the repaint procedure anyway, so it is not necessary to call it redundantly from here. See Chapter 6, *Handling Input*, for details about this.

In *canvas_event.c*, the `repaint_proc` simply clears the paint window and then displays the three messages in it, at a fixed position and using the default font. In the case of a pure repaint callback (from the Notifier, not the `event_proc`), the messages will just repeat the last event’s messages.

Example 5-3. The canvas_event.c program

```
/*
 *  canvas_event.c
 *  Demonstrates how to get keyboard and mouse events in an canvas
 *  window.  Looks for keyboards, pointer movement and button
 *  events and displays the info in the canvas.
 */
#include <X11/Xlib.h>
#include <xview/xview.h>
#include <xview/canvas.h>
#include <xview/xv_xrect.h>

void    event_proc(), repaint_proc();
char    kbd_msg[128], ptr_msg[128], but_msg[128];

/*
 * main()
 *      Create a canvas specifying a repaint procedure.
 *      Get the paint window for the canvas and set the input
 *      mask and the event procedure.
 */
main(argc, argv)
int argc;
char *argv[];
{
    Frame       frame;
    Canvas      canvas;

    /* Initialize XView */
    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    /* Create windows -- base frame and canvas. */
    frame = (Frame)xv_create(NULL, FRAME, NULL);

    canvas = (Canvas)xv_create(frame, CANVAS,
        XV_WIDTH,               300,
        XV_HEIGHT,              110,
        CANVAS_X_PAINT_WINDOW,  TRUE,
        CANVAS_REPAINT_PROC,    repaint_proc,
        NULL);
    window_fit(frame);

    /* Set input mask */
    xv_set(canvas_paint_window(canvas),
```

Example 5-3. The canvas_event.c program (continued)

```
        WIN_EVENT_PROC,         event_proc,
        WIN_CONSUME_EVENTS,
            KBD_DONE, KBD_USE, LOC_DRAG, LOC_MOVE, LOC_WINENTER,
            LOC_WINEXIT, WIN_ASCII_EVENTS, WIN_MOUSE_BUTTONS,
            NULL,
        NULL);

    /* Initial messages */
    strcpy(kbd_msg, "Keyboard: key press events");
    strcpy(ptr_msg, "Pointer: pointer movement events");
    strcpy(but_msg, "Button: button press events");

    /* Start event loop */
    xv_main_loop(frame);
}

/*
 * event_proc()
 *      Called when an event is received in the canvas window.
 *      Updates the keyboard, pointer and button message strings
 *      and then calls repaint_proc() to paint them to the window.
 */
void
event_proc(window, event)
Xv_Window window;
Event    *event;
{
    if (event_is_ascii(event))
        sprintf(kbd_msg, "Keyboard: key '%c' %d pressed at %d,%d",
                event_action(event), event_action(event),
                event_x(event), event_y(event));
    else
        switch (event_action(event)) {
            case KBD_USE:
                sprintf(kbd_msg, "Keyboard: got keyboard focus");
                break;
            case KBD_DONE:
                sprintf(kbd_msg, "Keyboard: lost keyboard focus");
                break;
            case LOC_MOVE:
                sprintf(ptr_msg, "Pointer: moved to %d,%d",
                    event_x(event), event_y(event));
                break;
            case LOC_DRAG:
                sprintf(ptr_msg, "Pointer: dragged to %d,%d",
                    event_x(event), event_y(event));
                break;
            case LOC_WINENTER:
                sprintf(ptr_msg, "Pointer: entered window at %d,%d",
                    event_x(event), event_y(event));
                break;
            case LOC_WINEXIT:
                sprintf(ptr_msg, "Pointer: exited window at %d,%d",
                    event_x(event), event_y(event));
                break;
            case ACTION_SELECT:
            case MS_LEFT:
```

Example 5-3. The canvas_event.c program (continued)

```
                sprintf(but_msg, "Button: Select (Left) at %d,%d",
                    event_x(event), event_y(event));
                break;
            case ACTION_ADJUST:
            case MS_MIDDLE:
                sprintf(but_msg, "Button: Adjust (Middle) at %d,%d",
                    event_x(event), event_y(event));
                break;
            case ACTION_MENU:
            case MS_RIGHT:
                sprintf(but_msg, "Button: Menu (Right) at %d,%d",
                    event_x(event), event_y(event));
                break;
            default:
                return;
        }

    /* call repaint proc directly to update messages */
    repaint_proc((Canvas)NULL, window,
        (Display *)xv_get(window, XV_DISPLAY),
        xv_get(window, XV_XID), (Xv_xrectlist *) NULL);
}

/*
 * repaint_proc()
 *      Called to repaint the canvas in response to damage events
 *      and the initial painting of the canvas window.
 *      Displays the keyboard, pointer and button message strings
 *      after erasing the previous messages.
 */
void
repaint_proc(canvas, paint_window, dpy, xwin, xrects)
Canvas        canvas;           /* Ignored */
Xv_Window     paint_window;     /* Ignored */
Display      *dpy;
Window        xwin;
Xv_xrectlist *xrects;           /* Ignored */
{
    GC gc = DefaultGC(dpy, DefaultScreen(dpy));

    XClearWindow(dpy, xwin);
    XDrawString(dpy, xwin, gc, 25, 25, kbd_msg, strlen(kbd_msg));
    XDrawString(dpy, xwin, gc, 25, 50, ptr_msg, strlen(ptr_msg));
    XDrawString(dpy, xwin, gc, 25, 75, but_msg, strlen(but_msg));
}
```

The result produced by Example 5-3 is shown in Figure 5-6.

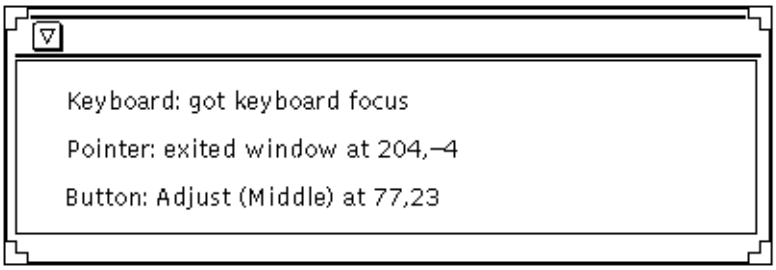

Figure 5-6. A window created with canvas_event.c

5.8 Canvas and Openwin Package Summaries

Table 5-1 shows the attributes for the `CANVAS` package and Table 5-2 shows the `OPENWIN` attributes. The macros for these packages are shown below. This information is described fully in the *XView Reference Manual*.

```
CANVAS_EACH_PAINT_WINDOW()
CANVAS_END_EACH
OPENWIN_EACH_VIEW()
OPENWIN_END_EACH()
```

Table 5-1. Canvas Attributes

CANVAS_AUTO_EXPAND	CANVAS_PAINTWINDOW_ATTRS
CANVAS_AUTO_SHRINK	CANVAS_REPAINT_PROC
CANVAS_FIXED_IMAGE	CANVAS_RESIZE_PROC
CANVAS_HEIGHT	CANVAS_RETAINED
CANVAS_MIN_PAINT_HEIGHT	CANVAS_VIEW_CANVAS_WINDOW
CANVAS_MIN_PAINT_WIDTH	CANVAS_VIEW_PAINT_WINDOW
CANVAS_NO_CLIPPING	CANVAS_VIEWABLE_RECT
CANVAS_NTH_PAINT_WINDOW	CANVAS_WIDTH
CANVAS_PAINT_CANVAS_WINDOW	CANVAS_X_PAINT_WINDOW
OPENWIN_VIEWCLASS	

Table 5-2. Openwin Attributes

OPENWIN_ADJUST_FOR_HORIZONTAL_SCROLLBAR	OPENWIN_SPLIT_DESTROY_PROC
OPENWIN_ADJUST_FOR_VERTICAL_SCROLLBAR	OPENWIN_SPLIT_DIRECTION
OPENWIN_AUTO_CLEAR	OPENWIN_SPLIT_INIT_PROC
OPENWIN_HORIZONTAL_SCROLLBAR	OPENWIN_SPLIT_POSITION
OPENWIN_NO_MARGIN	OPENWIN_SPLIT_VIEW
OPENWIN_NTH_VIEW	OPENWIN_SPLIT_VIEW_START
OPENWIN_NVIEWS	OPENWIN_VERTICAL_SCROLLBAR
OPENWIN_SHOW_BORDERS	OPENWIN_VIEW_ATTRS
OPENWIN_SPLIT	
WIN_COLUMNS	WIN_ROWS